Towards a Framework for Dedicated Operating Systems Development in High-End Computing Systems*

Jean-Charles Tournier †
Patrick G. Bridges,
Arthur B. Maccabe and
Patrick M. Widener
Dpt. of Computer Science
University of New Mexico
Albuquerque, NM 87131-0001

Zaid Abudayyeh‡,
Ron Brightwell and
Rolf Riesen
Sandia National Laboratories
PO Box 5800; MS 11110
Albuquerque, NM 87185-1110

Trammel Hudson§
Operating Systems Research, Inc.
1729 Wells Drive NE
Albuquerque, NM 87112

ABSTRACT
In the context of high-end computing systems, general-purpose operating systems impose overhead on the applications they support due to unneeded services. Although dedicated operating systems overcome this issue, they are difficult to develop or adapt. In this paper, we propose a framework, based on the *component* programming paradigm, which supports the development and adaptation of such operating systems. This framework makes possible the *a la carte* construction of operating systems which provide specific high-end computing system characteristics.

1. INTRODUCTION
In the context of high-end computing (HEC) systems, two main classes of operating systems are used, namely general purpose (e.g. Linux [11], K42 [2]) and dedicated ones (e.g. BlueGene/L [9]). General purpose operating systems provide a wide range of services and enable sophisticated applications with, for example, capabilities for visualization and inter-networking. However, this generality comes at the cost of performance for all applications that use the operating system because of the overheads of unnecessary services. Several studies have demonstrated and evaluated these overheads. As an example, [6] showed that a dedicated operating system for high performance systems transfers messages between nodes from 4 to 10 times faster than Linux does. However, the development of a dedicated operating system is complex and painful to achieve, as diverse characteristics such as application requirements, hardware specificities or associated programming models must be addressed.

Component-based software engineering (CBSE) appears to be a promising solution for the development of such operating systems. Indeed, one of the main claims of CBSE is to offer an easier way to build complex software by simply assembling software entities called components [18]. Moreover, a common characteristic of component models is their explicit specification of provided and required services, allowing the validation, from a functional point of view, a given composition of components.

Several recent projects (e.g. TinyOS [17], VEST [21], Koala [25] or Think [24]) have defined and developed component-based operating systems in order to build dedicated operating systems for applications with minimum effort. However, none of these projects address the specific requirements and characteristics of HEC.

In this paper, we present a component-based approach for developing dedicated HEC operating systems. We base our approach on the Fractal generic component model [7], enhancing and adapting Fractal to the specific requirements of HEC. The resulting component model allows us to construct dedicated OSs adapted to application needs, hardware characteristics, and the associated programming model.

The rest of this paper is organized as follows: section 2 identifies the motivations and the requirements for dedicated operating systems for HEC. Section 3 presents our component-based model which addresses those requirements. Section 4 presents some related works, while section 5 concludes the paper.

2. MOTIVATIONS AND REQUIREMENTS
In this section we present the motivations for dedicated operating systems for HEC and then identify the requirements of such operating systems.

2.1 Motivations
A general purpose operating system trades performance for generality, providing services that may not be necessary in HEC contexts. We illustrate this statement by comparing the performance of the `mg` and `cg` NAS B benchmarks on ASCI Red hardware [16] when running two different operating systems. We use Cougar, the productized version of the Puma operating system [26], and Linux as the general purpose operating system. To make the comparison as fair to

*This work was supported in part by Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

†{tournier,bridges,maccabe,widener}@cs.unm.edu
‡{zabuday,rrbright,rolf}@sandia.gov
§hudson@osresearch.net

Linux as possible, we ported the Cplant version of the Portals high-performance messaging [5] layer to the ASCI Red hardware. Cougar already utilizes this Portals for message transmission.

Linux outperforms Cougar on the `cg` benchmark with small number of nodes (less than 32) because Cougar uses older, less optimized compilers and libraries. As the number of nodes increases, application performance on Linux falls off. Similar effects happen with the `mg` benchmark, though `mg` on Cougar outperforms `mg` on Linux even on small numbers of nodes despite using older compilers and libraries. A variety of different overheads cause Linux's performance problems on larger-scale systems, including lack of contiguous memory layout (with associated TLB overheads) and suboptimal node allocations (due to limitations with Linux job-launch on ASCI Red).

Such operating system problems have also been seen in other systems. Researchers at Los Alamos, for example, have shown that excess services can cause dramatic performance degradations [20]. Similarly, researchers at Lawrence Livermore National Laboratory have shown that operating system scheduling problems can have a large impact on application performance in large machines [15].

2.2 Requirements

The previous section demonstrated the overheads imposed by general purpose operating systems. However, in order to build dedicated operating systems for HEC, we have to characterize how those systems may differ from instance to instance. Four main aspects have been identified: hardware, software, usage models, and environmental services. Finally, in order to take into account each of these aspects, we underline the properties needed to make an operating system dedicated.

2.2.1 Hardware evolution

The first aspect to be considered is the underlying architecture hardware of HEC systems. A wide variety of hardware architecture features are currently available, among them multiple processors, support for parcel-based processor-in-memory [22], multiple network interfaces, programmable network interfaces, and access to local storage. For example, the first terascale system, ASCI Red, is a traditional distributed memory massively parallel processing machine with thousands of nodes, each with a small number of processors. In contrast, the ASCI Blue Mountain machine was composed of 128-processor nodes, while ASCI White employs 16-way SMP nodes. The key challenge presented by different architectures is the need to take advantage of each of them by defining appropriate abstractions. The hardware architecture may also evolve dynamically since nodes can fail at run-time. This implies a global cooperation between each node to enforce the availability and serviceability of the overall system, which is highly dependent on the architecture. An operating system must also be flexible enough to integrate next generation hardware with a minimal impact. For example, the previously mentioned hardware architectures are expected to integrate multi-core or processor-in-memory chips in the near future.

2.2.2 Software evolution

HEC software characteristics are as varied as their hardware. Software is developed in the context of a particular programming model requiring a basic set of services. For example, in the explicit message passing model, data must be moved efficiently between local memory and the network. The same software may require extended functionality beyond the minimal set needed to support the programming model. Moreover, different interfaces to similar operating system services may be required. As an example, network packet reception may be signaled through interrupts for real-time or event-driven software, or by extending the kernel with specific handler codes for performance critical signals.

2.2.3 Usage models evolution

Two main classes of usage models are classically identified for HEC systems: capability-oriented and capacity-oriented usage. Capability-oriented usage is characterized by a small number of codes which rely on a relatively few operating system services. This usage model class implies a full-scale usage of the system and allows programmers to finely tune software using *hero* codes. Capability-oriented usage is usually further refined into restricted usage models such as *dedicated* or *space-shared*. The capacity-oriented class of usage models supports a wide range of different codes. To do this, it provides more complex services such as dynamic loading, shared libraries, and language run-time facilities. Capacity-oriented usage supports more flexible refined models such as *timesharing*. As large scale systems age, they frequently transition from specialized capability-oritented usage for a handful of applications to capacity usage for a wide range of applications. Their original operating systems must therefore be enhanced, with both high and low level services being added or modified.

2.2.4 Environmental services evolution

The variety of shared environmental services that operating systems must support, such as file systems and checkpointing, cannot be expected to remain constant. New implementations of these services are continually being developed, and these implementations require changing operating system support. Moreover, these services are often implemented at user-level in lightweight operating systems. In this case, the operating system must provide a way to authenticate trusted shared services to applications and other system nodes.

These hardware, software, usage model and environmental service characteristics lead to the need for a framework to build operating systems *a la carte* in order to develop dedicated operating systems. Each characteristic requires high flexibility from the operating system part in order to fit the system and thus maximize the overall system performance. We outline the requirements of such a framework below.

2.2.5 Operating system framework requirements

Configurability is a key aspect of dedicated operating systems, since it defines the way an operating system can be specialized to a given system. Thus, in order to be as dedicated as possible, an operating system has to take into account several aspects of configuration, namely depth, granularity and time.

The depth aspect defines the lowest level of the system that can be configured. Unlike most of the classical approach, we want full configurability. As an example, device drivers or the scheduler may be replaced, but we are also concerned with lower level functions such as memory page table management or process state saving. It should be possible to configure an operating system which runs a single application in a flat address space as well as a multitasking system that runs on multiple CPUs. The latter case has requirements (such as processor synchronization) which do not apply in the first case; the systems designed for the former case should be able to exclude synchronization for performance reasons. In other words, we do not want to predefine a fixed set of core functionalities.

Granularity identifies the size of the software pieces that can be configured. Granularity is usually a tradeoff between configurability and performance. The smaller the granularity the more dedicated the system can be dedicated, but the overhead of coordinating more, smaller components is higher. Thus, we do not want to limit granularity as the needs of each part of the system are different. For example, a MMU is a relatively large unit of configuration, while synchronization mechanisms such as mutexes are much smaller.

Time of configuration identifies the moment when the operating system may be dedicated to a specific one of the possible system configurations. Two main times of configuration are usually identified: static (i.e. development stage) and dynamic (i.e. execution stage). In order to maximize the continuity of service for HEC systems and applications, we want to be able to reconfigure an operating system dynamically.

3. APPROACH

Our approach is based on the component concept, which reduces the effort necessary to build a system. Each part of the system is identified as a component, and they are composed to form a complete operating system. We chose Fractal [7] and its associated framework called Think [10, 24], as our component system. Several reasons motivated this choice. First, from a model point of view, Fractal provides a minimal and simple generic component model which can be easily extended or adapted. Since no model addresses HEC requirements exclusively, an existing model that is easily extended is beneficial. Secondly, Think provides a complete tool chain for composing operating system components. Although Think originally targeted embedded systems, its conformance to the Fractal model allows us to reuse it for HEC. The systematic use of components makes Think fully configurable and adaptable, and allows a fine grain control over each part of a system (especially resources). Finally, the framework does not predefine the granularity of a component since underlying components are hierarchical. The first part of this section provides an overview of Fractal and the Think framework, while the second part focuses on the extension applied to the model to deal with HEC specific requirements. The last part illustrates the proposed approach through an example.

3.1 A component-based approach

The Fractal model aims to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. It promotes a component approach and defines a component model made of five key concepts: component, content, controller, interface and binding. A component is a run-time entity and is made of a content and a controller. A controller aims to control the content of the component (e.g. life-cycle or configuration), while a content implements the functionality. This distinction enforces the *separation of concern* principle since the content focuses on the functional part while the controller focuses on the non-functional one. A component has an arbitrary size and a content may be made of other components, enabling hierarchical composition. The hierarchical recursion ends at primitive components which are defined by content using an implementation language (e.g. C or assembly code) rather than a configuration of other components. An interface is an access point to a component that supports a finite set of methods. Interfaces can be either *server*, which corresponds to access points accepting incoming method calls, or *client*, which corresponds to access points accepting outgoing method calls. Moreover server interfaces have a given multiplicity, which specifies an upper bound (1,n,infinite) on the number of client interfaces they can be bound to. Finally, communication between components is only possible if their interfaces are bound. A binding is an oriented connection between client and server interfaces and can only be set if the methods defined by the client interfaces are a subset of the ones defined by the server interface.

The Think framework aims to build configurable embedded operating systems. It implements the Fractal component model, which is systematically applied to build a system. Think provides a set of tools to define, describe and compose components, as well as a library of components. Think defines an Interface Description Language (IDL) in order to allow local or remote interoperation of components, as well as an Architecture Description Language (ADL) that describes a system configuration. The IDL is a subset of Java and is used to define component interfaces. The ADL is used to describe each individual component and their different composition and bindings. The Think tool chain includes the IDL and ADL compilers as well as an offline configurator which creates operating system images by assembling components (no facilities for dynamic configuration are available). The component library is mainly divided in two parts, the hardware dependent layer and classical operating system services. The hardware dependent components reify exceptions and provide memory management and a set of device drivers. The existing set of components has been mainly developed for ARM and PowerPC processors. The classical operating system service components implement various services such as memory organization (e.g. flat or paged memory), thread components, scheduling (e.g. round-robin, priority) or networking (e.g. TCP/IP protocol stack).

Experiments using Think demonstrate that component based development is not necessarily less efficient. The main cost engendered by components is the memory footprint overhead[1] that has been estimated to an average of 2% (overhead increasing as average component size decreases). How-

[1]The memory overhead is caused by system component structure maintained during execution in order to allow dynamic management of components.

ever, the component approach allows us to master operating system complexity and enables a fine grain control over resources, since each part of the system is reified as a component.

3.2 Components for HEC

Think has limitations with respect to specific requirements for HEC systems. As an example, Think only provides the classical function call communication pattern, while event-based communication is required for HEC. Another lack of the Think framework is its limited component library concerning HEC needs such as memory management. Due to space limitations, we present in this section only our proposed extension for event communication.

Although the classical function call is a natural programming model, it is not easy to split across execution domains because of the synchronization frequently implicit in it (e.g return values). Events are usually more difficult to program with. However, they are easier to split and manage across execution contexts since they are naturally asynchronous and do not return values.

The main goal of the event extension is to provide a simple and efficient way to program with events. The extension has to be as open as possible, avoiding predefinition of the event semantic and communication pattern. To achieve these goals, events are reified as components. Events therefore can be managed as classical components, including their life cycle (e.g. start, pause or stop events) or attributes. Think event components can be serialized when consumers and producers are distributed. The extension implementation provides two kind of events: the first one implements a synchronous semantic (a producer waits for all consumers to consume the event), while the second one provides an asynchronous semantic (producers do not wait, and events are placed in a queue once they are generated). Each type can be applied to various communication patterns. The implementation provides the *mediator* communication pattern (when several consumers are registered to a single event) as well as direct communication. In this way, the extension avoids the cost of the mediator pattern when only one producer and one consumer are present. In order to describe the event communication between components, the event extension enhances the original Think ADL, adding key words to describe which event is produced/consumed by components and to select a communication pattern.

Although the event extension simplifies event-based programming, it has a cost, mainly in memory consumption. The binary form of an asynchronous event in the PowerPC Think implementation is less than 2 kB. Note that if the event is just a fixed integer without any need of management, it does not have to be implemented by a component. The current implementation requires events with similar management requirements (e.g. life-cycle) and semantics to be bundled in a single component.

3.3 Example

We illustrate the proposed approach using a componentized version of Puma [26], a lightweight operating system for massively parallel systems developed at Sandia National Laboratories. Puma is a message passing kernel based on the concept of portals, which are openings in the address space of an application process. Puma has three main parts: the quintessential kernel (Q-Kernel), the process control thread (PCT) and the application processes (AP). The Q-Kernel is responsible for controlling access to the physical resources provided by a processor node. More precisely, it is in charge of communication facilities and address space protection. The PCT provides process management functions (e.g process creation and scheduling) and manages access to the physical resources. Thus, control mechanisms and management policies are separated. Finally, the application processes are created and scheduled by the PCT and have access to the hardware through the Q-Kernel. Moreover, application processes communicate through portals, placing messages directly into the memory of receivers. Different types of portals are defined, distinguished by the management policy associated with the portal memory.

A natural way to componentize Puma is to keep the original design. Accordingly, the Q-Kernel, PCT, and application processes are all reified into component types. The Q-Kernel component provides several interfaces, which are referred as the kernel entry points in the original design. Each entry point is either used by the PCT, the application processes or both. In order to enforce separation between the entry points, the Q-Kernel component provides one interface dedicated to the PCT, one to the application processes and a last one for both of them. Moreover, the Q-Kernel is made of several sub-components. These are the context component, which provides information about the current running contexts; the out-going messages component, which is in charge of sending messages; the interrupt component, which reifies the different possible interrupts and exceptions; and finally the Q-logic component. Q-logic implements the logic of the Q-Kernel and requires services from the other sub-components. Figure 1 illustrates the Q-Kernel component.

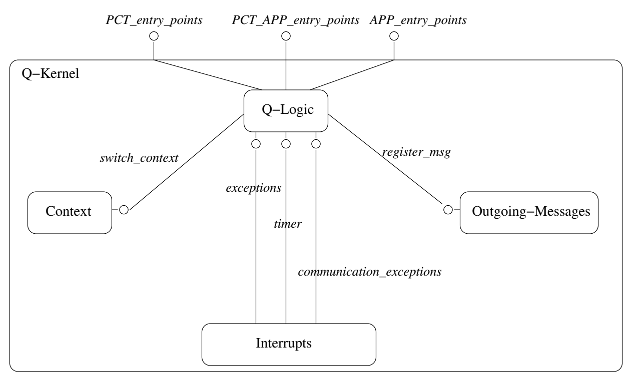

Figure 1: Design of the Q-Kernel component.

The PCT component has two required interfaces in order to communicate with the Q-Kernel component via the entry-points. Moreover, it provides interfaces in order to create and schedule application processes components. It also requires several interfaces from the application process component in order to check their mail-boxes. The PCT and the application processes exchange requests through mail-boxes owned by each process. The PCT, unlike the Q-Kernel, is not composed of sub-components.

While, the PCT and Q-Kernel are singletons, the AP component may be instantiated several times. An AP component requires one interface to the Q-Kernel (for Q-Kernel entry-points) and provides one interface to the PCT (for mail-box check). Moreover, an AP is made of four different components. The first one, called process-logic component, implements the logic of the of the application process; the others are dedicated to the different task of managing a process in Puma. Separation of concerns is enforced by separating the technical aspect (application component) to the non-technical one. The other AP component are a mail-box component, a signal component and a portal one. Moreover, it is worth noting that the portal component may have different implementations but still provides the same interface to the application component. Figure 2 illustrates the design of the application process component.

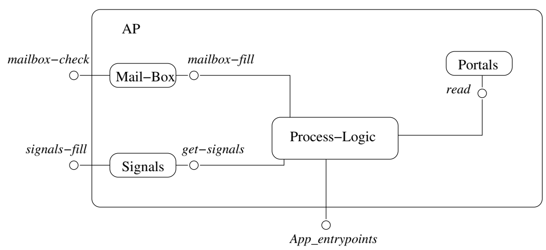

Figure 2: Design of the application process component.

4. RELATED WORKS

Several projects have tried to deal with configurability in operating systems. According to [23], we can classify these projects in as general purpose systems, dedicated systems and distributed systems. Although all these projects have the same main goal, their implementations and therefore their configuration possibilities vary.

In the area of general purpose systems, MetaOS [14], Spin [4], OSKit [12] and Linux [11] define mechanisms in order to be adapted to a given environment. However, these mechanisms are either too coarse grain (Linux, OSKit) to allow an efficient customization, or are based on costly mechanism such as the meta approach in MetaOS. In contrast, our approach provides arbitrary granularity to achieve efficient customization, but also addresses system invariants order to minimize configuration cost.

The area of dedicated systems is the most prolific domain for configurable operating systems. Projects such as Pebble [13], eCos [1], Scout [19] or TinyOS [17] provide tools for building dedicated operating systems for embedded systems. However, Pebble, eCos and Scout only provide coarse grain configuration and are moreover based on an underlying kernel which can not itself be reconfigured. TinyOS has a similar goal and approach to ours since it provides a full componentization of the operating system and allows event communication. However, TinyOS only provides configuration at development stage, preventing the flexibility needed by HEC applications.

In the area of distributed systems, the existing configurable operating systems again are not fine grain enough to allow efficient customization (e.g K42 [2], Choices [8]). Moreover, these projects pre-define the overall structure of an operating system, prohibiting migration between capability and capacity designs.

5. CONCLUSION AND FUTURE WORKS

In this paper, we have presented an argument for a framework for building dedicated operating systems in high-end computing systems. Based on the results of preliminary experiments, we conclude that the demands of current and future high systems cannot be addressed by a general-purpose operating system, but rather by a dedicated one with adaptation capabilities. To address this problem, we propose an approach based on a component-based framework in order to build operating systems *a la carte*. Our approach extends an existing framework initially dedicated to embedded systems to capture the specific requirements of HEC systems. Our approach minimizes the overhead of unneeded features, and enables the construction of new operating systems adaptable to evolving demands and requirements.

We are currently working on multiple implementation efforts. One of our goals in so doing is to provide "subtrees" of components that can be used as-is to build operating system kernels for HEC systems. The first such subtree contains components for memory management. The Think framework contains a MMU hierarchy with implementations for PowerPC and the ARM processor. We are generalizing those interfaces to accomodate the Xen hypervisor memory management design, as a first step toward a functional implementation on top of Xen [3]. Other efforts include the reengineering the Puma operating system with our proposed approach as described in section 3.3.

6. REFERENCES

[1] http://sources.redhat.com/ecos/.

[2] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. *IBM Systems Journal*, pages 21–27, 2004.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[5] R. Brightwell, T. Hudson, R. Riesen, and A. Maccabe. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, 1999.

[6] Ron Brightwell, Rolf Riesen, Keith Underwood, Trammell Hudson, Patrick Bridges, and Arthur B.

Maccabe. A Performance Comparison of Linux and a Lightweight Kernel. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster2003)*, 2003.

[7] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, and JB. Stefani. An Open Component Model and its Support in Java. In *7th Int. Symp. CBSE*, 2004.

[8] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing choices:an object-oriented system in c++. *Communications of the ACM*, 36(9):117–126, 1993.

[9] N. R. Adiga et al. An Overview of the BlueGene/L Supercomputer. In *2002 ACM/IEEE Conference Supercomputing*, 2002.

[10] J.F. Fassino, J.B. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *USENIX 2002 Annual Conference*, 2002.

[11] Linux for High Performance Computing. http://www.linuxhpc.org/.

[12] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: a substrate for kernel and language research. *SIGOPS Oper. Syst. Rev.*, 31(5):38–51, 1997.

[13] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The pebble component-based operating system. In *USENIX Techincal Conference*, pages 267–282, 1999.

[14] Michael Horie, James C. Pang, Eric G. Manning, and Gholamali C. Shoja. Designing meta-interfaces for object-oriented operating systems. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 989–992, 1997.

[15] T. Jones, W. Tuel, L. Brenner, J. Frier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operatin system. In *SC*, 2003.

[16] Sandia National Laboratories. ASCI Red, http://www.sandia.gov/ASCI/TFLOP.

[17] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. *Ambient Intelligence*, chapter TinyOS: An Operating System for Wireless Sensor Networks. Springer, 2005.

[18] B. Meyer and C. Mingis. Component-Based Development: from Buzz to Spark. In *IEEE Computer*, July 1999.

[19] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*, 1994.

[20] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC*, 2003.

[21] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *Ninth IEEE Real-Time and Embedded Technology and Applications Symposium*, page 58, 2003.

[22] T. L. Sterling and H. P. Zima. The gilgamesh MIND processor-in-memory architecture for petaflops-scale computing . In *International Symposium on High Performance Computing*, 2002.

[23] J.C. Tournier. A Survey of Configurable Operating Systems. Technical Report TR-CS-2005-43, University of New Mexico, Computer Science Department, 2005.

[24] J.C. Tournier and J.P. Fassino. The Think Components-Based Operating System. In Hermes Science and Lavoisier Company, editors, *Model Driven Engineering for Distributed Real-time Embedded Systems*. International Scientific and technical Encyclopedia, 2005.

[25] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.

[26] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: an operating system for massively parallel systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 56–65, 1994.